UNIT 3. DIVIDE AND CONQUER ALGORITHMS

1. Techniques for designing divide and conquer algorithms.

In the divide and conquer algorithms, it is a matter of breaking down the case to be solved into smaller subcases of the same problem. Subsequently each subcase is solved independently and the results are combined to build the original case solution. This process is usually applied recursively and the efficiency of this technique depends on how the subcases are solved. It is, therefore, a scheme in 3 stages:

1) Divide. Divide the original problem into smaller subproblems
2) Conquer. These subproblems are solved independently:
   - Directly if they are simple.
   - Reducing to simpler cases (typically recursively)
3) Combine. The partial solutions are combined to obtain the solution of the original problem.

The most frequent case corresponds to k = 2, that is, the problem is divided in half.

The pseudocode is as follows:

```
function DivideAndConquer (c: typecase) : typesolution;
        var
                x1,..., xk : typecase;
                y1,...,yk: typesolution;
        if c is simply enough then
                return solution_simple(c)
        else
                descompose c in x1, ..., xk
                for i← 1 to k do
                        yi ← DivideAndConquer (xi)
                efor
                return combine_solution_subcases(y1, ..., yk)
        eif
efunction
```

For the divide and conquer approach to be worthwhile, the following conditions must be met:

- It must be possible to decompose the case to solve in sub-cases.
- Recursive formulation never solves the same subproblem more than once.
- It must be possible to compose the solution from the solutions of the subcases in an efficient way.
- The subcases should be approximately of the same size.

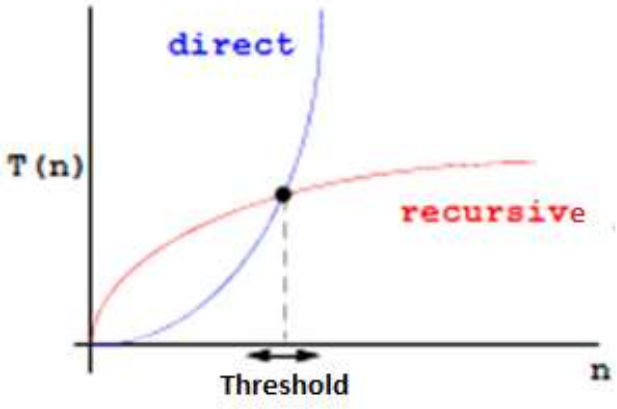
2. Determination of the threshold.

When the problem is small enough no further divisions are made. This limit is marked by the threshold. The threshold will be a n_0 such that when the size of the case to be treated is less or equal, it is solved with a basic algorithm, that is, no more recursive calls are

generated. The determination of the optimal threshold is a complex problem. Given a particular implementation, it can be calculated empirically. A hybrid technique can also be used for its calculation, which consists of the following:

1) Obtain the algorithm's recurrence equations (theoretical study).
2) Determine empirically the values of the constants that are used in those equations for the concrete implementation that is used.
3) The optimal threshold is estimated by finding the size n of the case for which there is no difference between directly applying the classical algorithm or moving to a deeper level of recursion.

In any case, the direct algorithm will be used when the size of the problem is smaller than the chosen threshold.



3. Computational cost.

Let n be the size of the original case and let k be the number of subcases: there exists a constant b such that the size of the k subcases is approximately n/b. If g(n) is the time required by the algorithm, divide and conquer in cases of size n, without counting the time necessary to make recursive calls, then:

$$t(n) = k*t(n/b)+g(n)$$

where t(n) is the total time required by the divide and conquer algorithm, provided n is large enough.

If there is an integer p such that $g(n) \in O(n^p)$, it can be concluded that:

$$T(n) = \begin{cases} O(n^p) & k < b^p \\ O(n^p \log n) & k = b^p \\ O(n^{\log_b k}) & k > b^p \end{cases}$$

4. Binary search.

Let T[1..n] be an ordered vector such that T[j]≥T[i] whenever n≥j≥i≥1 and let x be an element. The problem is to look for x in T. That is, we want to find an i such that n≥i≥1 and x=T[i], if x∈T.

The sequential search algorithm will be in the order of O(n):

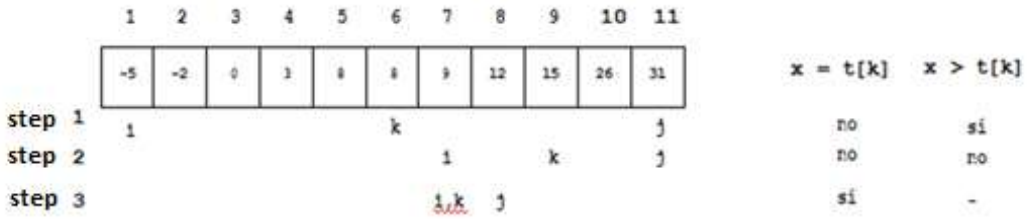
```
function SequentialSearch(integer[1..n] t, integer x)
        for i:=1 to n do
                if t[i] = x then
                        return i          // found
                eif
        efor
        return -1         // not found
efunction
```

Identification of the problem with the divide and conquer scheme:

1) Divide: The problem can be broken down into smaller subproblems (k = 1).
2) Conquer: There are no partial solutions, the solution is unique.
3) Combine: Not necessary.

It is one of the simplest applications of divide and conquer. It really does not consist on dividing the problem, but on reducing its size in each step. It is a divide and conquer algorithm of reduction or simplification.

Example: search for x = 9.



Steps of the binary search algorithm:

1) Compare x with the element that is in k=(i + j)/2
2) If T [k]≥x the search continues in T[i..k]
3) If T [k]<x the search continues in T[k + 1..j]

These steps will continue until you locate x, or determine that it is not in T. The pseudocode would be:

```
function BinarySearch(integer[1..n] t, integer i, integer j, integer x)
        // obtains the center
        k := (i + j)/2
        // direct case
        if i > j then
                return -1        // not found
        eif
        if t[k] = x then
                return k         // found
        eif
        // recursive case
        if x > t[k] then
                return BinarySearch (t, k+1, j, x)
        else
                return BinarySearch (t, i, k-1, x)
        eif
efunction
```

As can be seen, the time required by a divide and conquer algorithm is of the form:

$$t(n)=k*t(n/b)+g(n)$$

For this algorithm each call generates a recursive call (k=1), the size of the subproblem is half of the original problem (b=2) and without considering the recurrence the rest of the operations are O(1), then g(n) is $O(1) = O(n^0)$ (p=0). We are in the case $k = b^p$ ($1=2^0$), then t(n) is $O(n^p \log n)=O(n^0 \log n)=O(\log n)$.

5. Sort.

Given a T[1..n] vector, initially disordered, it will be ordered by applying the Divide and Conquer technique by splitting the initial vector into two smaller subvectors. Two techniques will be used:

- Mergesort
- Quicksort

5.1. Mergesort.

Algorithm steps:

1. Divide the vector into two halves
2. Those two halves are ordered recursively
3. They merge into a single ordered vector.

In order to apply this algorithm it is necessary an efficient procedure to merge two ordered matrices that would be as follows:

```
procedure fuse(U[1..m+1],V[1..n+1],T[1..m+n])
        {Fuse the ordered matrices U[1..m+1] and V[1..n+1] storing them on
        T[1..m+n]. U[1..m+1] y V[1..n+1] are used as sentinels}
        i,j:=1
        U[m+1],V[n+1]:=∞
        for k=1 to m+n do
                if U[i] <V[j] then
                        T[k]=U[i]
                        i=i+1
                else
                        T[k]=V[j]
                        j=j+1
```

Once we have this procedure the algorithm of order by fusion will be:

```
procedure mergesort(T[1..n])
        if n is small enought then sort(T)
        else
                U[1..n/2]=T[1..n/2]
                V[1..n/2]=T[1+n/2..n]
                mergesort(U[1..n/2])
                mergesort(V[1..n/2])
                fuse(U,V,T)
```

Computational cost:

$$T(n) = \begin{cases} O(n^p) & k < b^p \\ O(n^p \log n) & k = b^p \\ O(n^{\log_b k}) & k > b^p \end{cases}$$

In mergesort k=2 (two subproblems), b=2 (problems of half size) and p=1 (the complexity of ordering the simple case is O(n)), then $k=b^p$ and, therefore, the problem is O(nlogn)

5.2. Quicksort.

In this case, an element of the matrix to be ordered is selected as a pivot and the vector is divided on both sides of the pivot. The elements greater than the pivot are stored to its right and the others to its left. The vector is ordered by recursive calls to this algorithm. When selecting the pivot any element is valid, so it is normal to choose the first element of the vector, although the optimal pivot is the median of the elements of the vector.

In the quicksort algorithm an important part is to place the pivot in the appropriate position for which this procedure is used:

```
procedure pivot(T[1..j],var b)
        {Change the elements of the matrox T[i..j] and provides the value b such
        as, at the end i≤b≤j; T[k]≤p for all i≤k<b, T[b]=p and T[k]>p for all b<k≤j
        where p is the initial value of T[i]}
        p=T[i]
        k=i
        b=j+1
        repeat k=k+1 until T[k]>p or k≥j
        repeat b=b-1 until T[b]≤p
        while k<b do
                interchange T[k] and T[b]
                repeat k=k+1 until T[k]>p
                repeat b=b-1 until T[b]≤p
        interchange T[i] and T[b]
```

With this procedure the ordering algorithm would be:

```
procedure quicksort(T[i..j])
        {Order the submatrix T[i..j] in an not decreasing order}
        if j-i is small enought then sort (T[i..j])
        else
                pivot(T[i..j],b)
                quicksort(T[i..l-1])
                quicksort(T[l+1..j])
```

To calculate the cost it is assumed that the subproblems are balanced and, in that case, the cost is O(nlogn).

6. Power of integers.

Once given the positive integers a and n, it is about calculating a^n. A naive solution would be:

```
function pot0(a,n)
        r:=1
        for i:=1 to n do r:=r*a
        return r
```

Let a and n be 2 integers. If m is the size of a, the operation an, if the classical algorithm of multiplication is used, the cost is polynomial: $O(m^2n^2)$, while using algorithm divide and conquer, the cost is reduced to: $O(m^{log3}n^2)$.

However, there is a more efficient algorithm for the power of integers, considering that:

$$x^{25} = (((x^2x)^2)^2)^2x$$

Taking this into account, an algorithm for potentiation of type divide and conquer can be written

```
function expoDV(a,n)
        if n=1 then return a
        if n even then return[expoDV(a,n/2)]^2
        return a*expoDV(a,n-1)
```

7. Multiplication of matrices.

Let A and B be two matrices of size n x n, you want to calculate your product by applying the divide and conquer method. Two posibilities:

- When n is power of 2
- When n is not power of 2: Add rows and columns of zeros until it is.

Example:

A	1	2	3	4
1	8	4	5	3
2	3	8	4	6
3	6	1	3	3
4	7	2	7	5

X

B	1	2	3	4
1	6	2	3	6
2	2	3	9	5
3	5	4	6	3
4	4	5	2	1

=

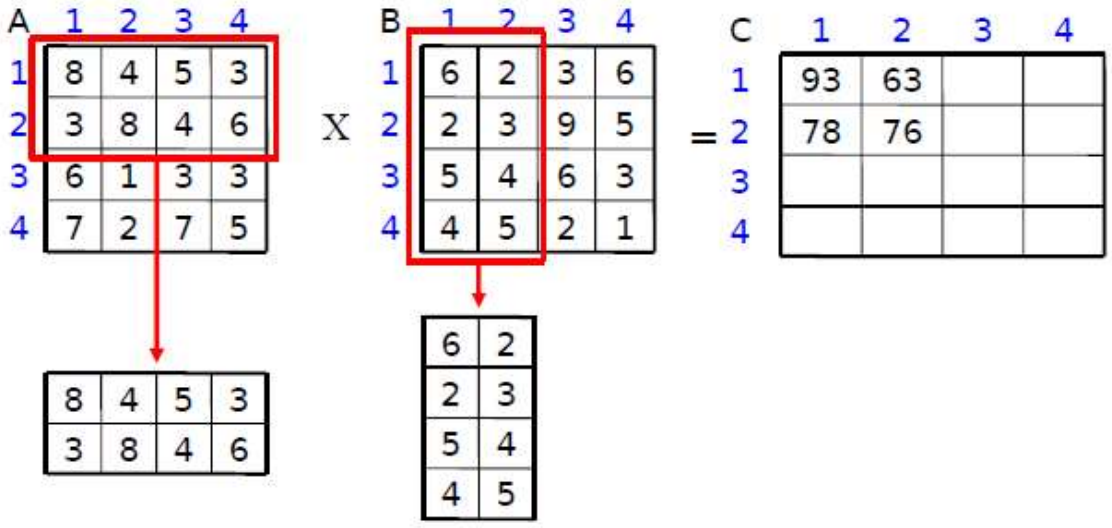
C	1	2	3	4
1	93	63		
2	78	76		
3				
4				

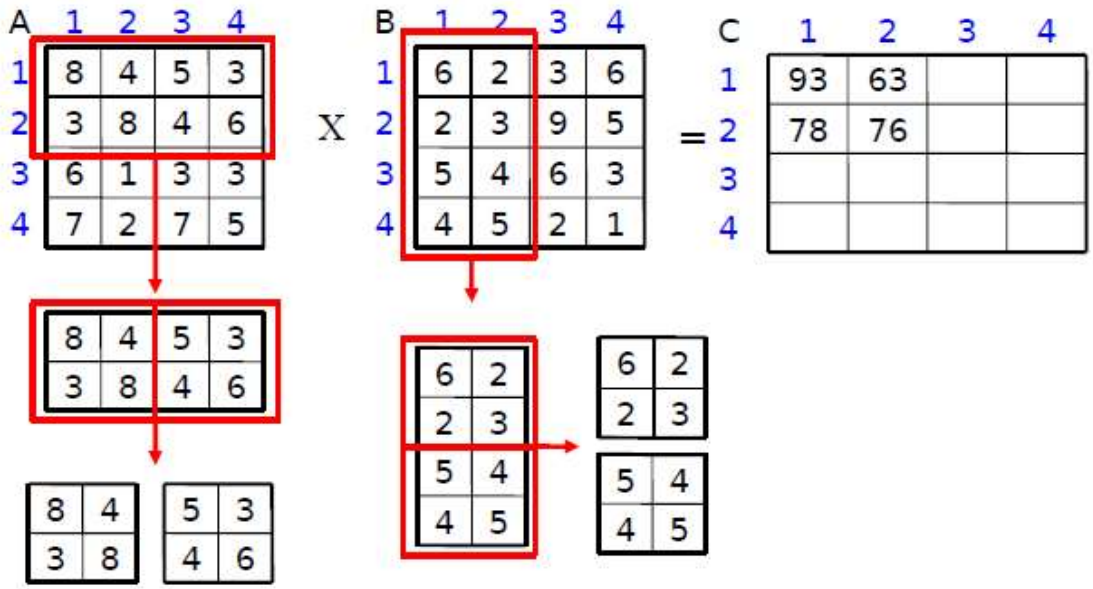
$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31} + A_{14} * B_{41}$$

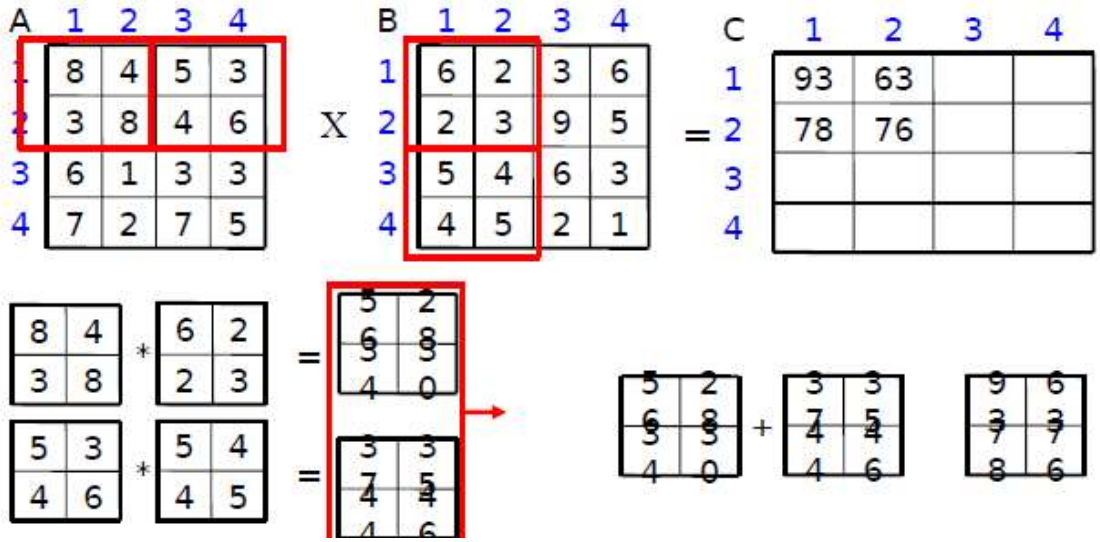
$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32} + A_{14} * B_{42}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21} + A_{23} * B_{31} + A_{24} * B_{41}$$

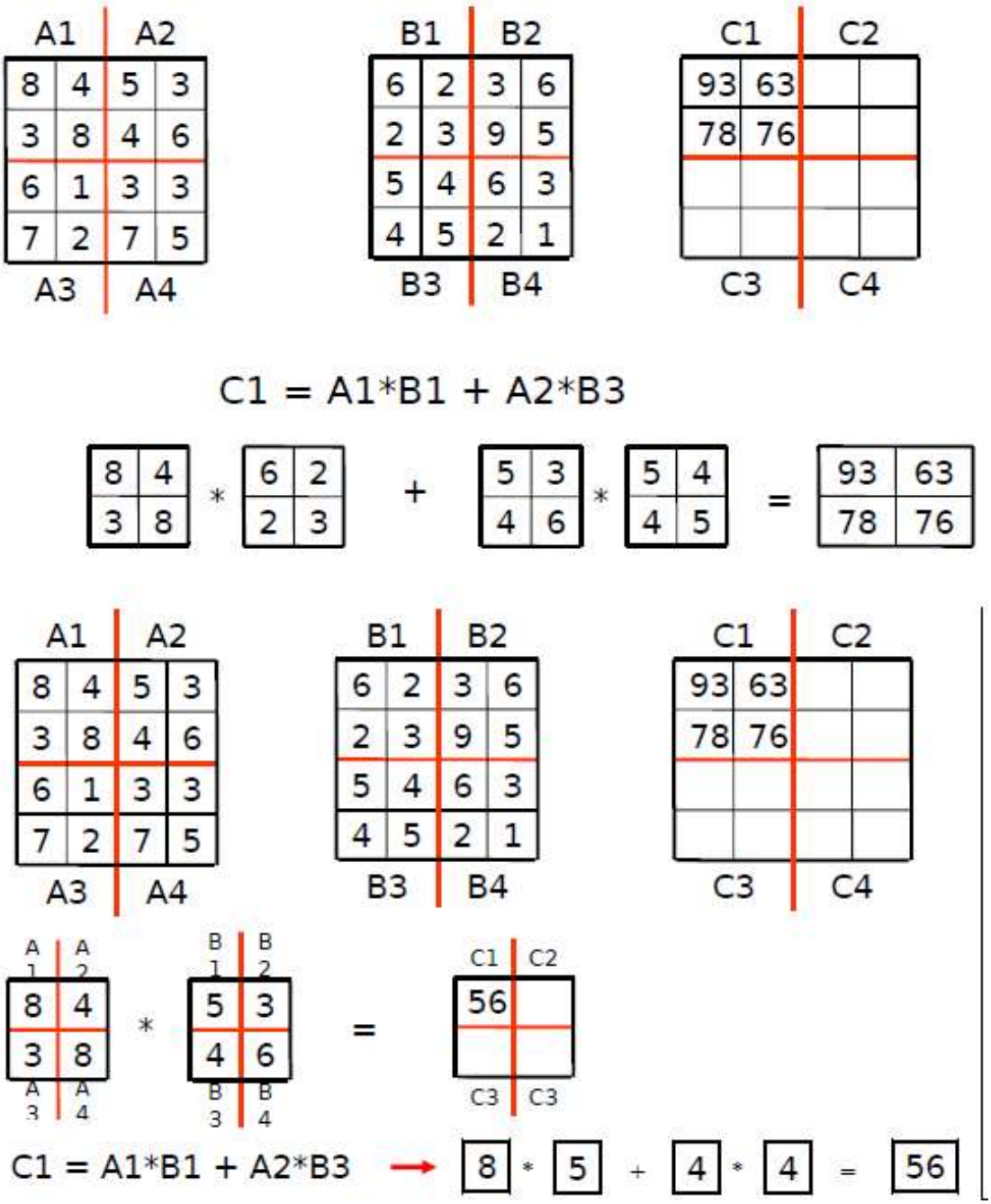
$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22} + A_{23} * B_{32} + A_{24} * B_{42}$$




A 1 2 3 4
1 8 4 5 3
2 3 8 4 6
3 6 1 3 3
4 7 2 7 5
X
B 1 2 3 4
1 6 2 3 6
2 2 3 9 5
3 5 4 6 3
4 4 5 2 1
=
C 1 2 3 4
1 93 63
2 78 76
3
4


A 1 2 3 4
1 8 4 5 3
2 3 8 4 6
3 6 1 3 3
4 7 2 7 5
X
B 1 2 3 4
1 6 2 3 6
2 2 3 9 5
3 5 4 6 3
4 4 5 2 1
=
C 1 2 3 4
1 93 63
2 78 76
3
4

8 4
3 8
*
6 2
2 3
=
5 3
4 6
*
5 4
4 5
=
+



Identification of the problem with the divide and conquer scheme:

- Division: The problem can be broken down into smaller subproblems (k=4).
- Conquer: Submatrices continue to divide until they reach size 1.
- Combine: Add the intermediate results

The pseudocode would be:

```
procedure multiply(A,B:typematrix; var C:typematrix)
var A11,A12,A21,A22, B11,B12,B21,B22, C11,C12,C21,C22: typematrix
        if size(A)=1 then
                C=A*B
        else
                Divideinquarters(A,A11,A12,A21,A22)
                Divideinquarters (B,B11,B12,B21,B22)
                for i=1 to 2 do
                        for j=1 to 2 do
                                Initialize(Cij)
                                for k=1 to 2 do
                                        multiply(Aik,Bkj,Caux)
                                        sum(Cij,Caux)
                                efor
                        efor
                efor
                Placequarters(C,C11,C12,C21,C22)
        eif
eprocedure
```